第3章 栈和队列

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

第3章 栈和队列

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

第3章 栈和队列

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

第3章 栈和队列

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

第3章 栈和队列

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

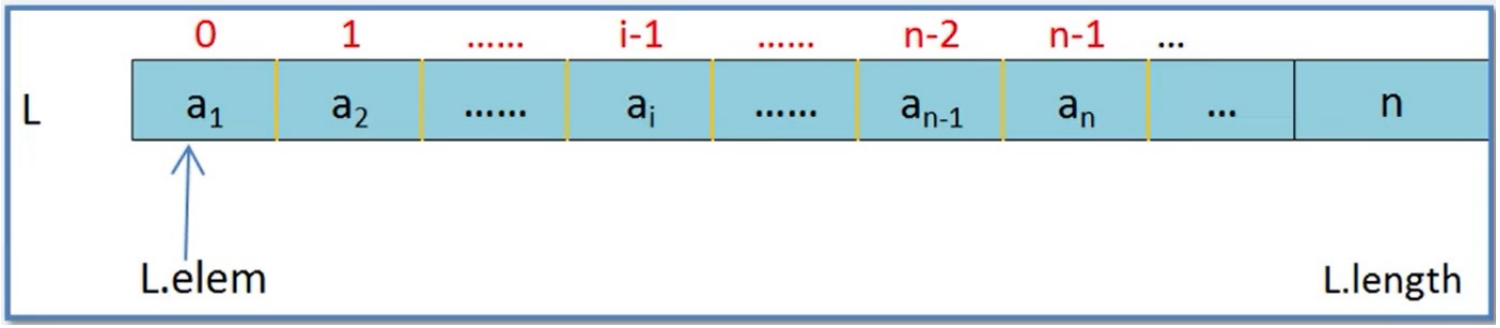
3.1 栈和队列的定义和特点

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

- 栈和队列是两种常用的、重要的数据结构
- 栈和队列是限定插入和删除只能在表的“端点”进行的线性表

3.1 栈和队列的定义和特点

- 普通线性表的插入和删除操作



插入第i个位置

删除第i个位置

3.1 栈和队列的定义和特点

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

- 栈和队列是两种常用的、重要的数据结构
- 栈和队列是限定插入和删除只能在表的“端点”进行的线性表

线性表	栈
Insert(L, i x) 1<=i<=n+1	Insert(S, n+1, x)
Delete(L, i) 1<=i<=n	Delete(S, n)

3.1 栈和队列的定义和特点

栈——后进先出

栈的应用

□ 由于栈的操作具有后进先出的固有特性，使得栈成为程序设计中的有用工具。另外，如果问题求解的过程具有“后进先出”的天然特性的话，则求解的算法中也必然需要利用“栈”。

- 数值转换
- 括号匹配的检验
- 行编辑程序
- 迷宫求解
- 表达式求值
- 八皇后问题
- 函数调用
- 递归调用的实现

3.1 栈和队列的定义和特点

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

- 栈和队列是两种常用的、重要的数据结构
- 栈和队列是限定插入和删除只能在表的“端点”进行的线性表
- 栈和队列是线性表的子集（是插入和删除位置受限的线性表）

线性表	栈	队列
Insert(L, i x) 1<=i<=n+1	Insert(S, n+1, x)	Insert(Q, n+1, x)
Delete(L, i) 1<=i<=n	Delete(S, n)	Delete(Q, 1)

3.1 栈和队列的定义和特点

队列——先进先出

队列的常见应用

- 由于队列的操作具有先进先出的特性，使得队列成为程序设计中解决类似排队问题的有用工具。
  - 脱机打印输出：按申请的先后顺序依次输出
  - 多用户系统中，多个用户排成队，分时地循环使用CPU和主存
  - 按用户的优先级排成多个队，每个优先级一个队列
  - 实时控制系统中，信号按接受的先后顺序依次处理
  - 网络电文传输，按到达的时间先后循序依次进行

3.1 栈和队列的定义和特点

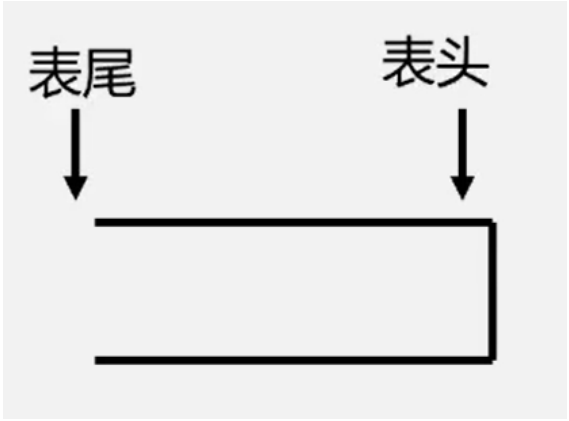
杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

- 栈和队列是两种常用的、重要的数据结构
- 栈和队列是限定插入和删除只能在表的“端点”进行的线性表
- 栈和队列是线性表的子集（是插入和删除位置受限的线性表）

线性表	栈	队列
Insert(L, i x) 1<=i<=n+1	Insert(S, n+1, x)	Insert(Q, n+1, x)
Delete(L, i) 1<=i<=n	Delete(S, n)	Delete(Q, 1)

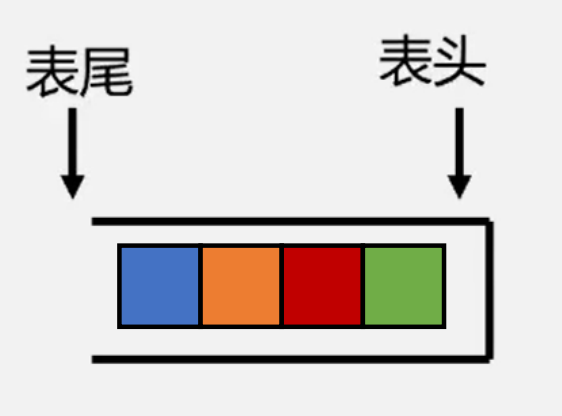
3.1.1 栈的定义和特点

- 栈（stack)是一个特殊的线性表，是限定仅在一端（通常是表尾）进行插入和删除操作的线性表。



3.1.1 栈的定义和特点

- 栈（stack)是一个特殊的线性表，是限定仅在一端（通常是表尾）进行插入和删除操作的线性表。
- 又称为后进先出（Last In First Out)的线性表，简称LIFO结构



3.1.1 栈的定义和特点

- 栈的相关概念

栈是仅在表尾进行插入、删除操作的线性表。

表尾（即a_n 端）称为栈顶Top；表头（即a_1 端）称为栈底Base

例如：栈 S = （a_1，a_2，a_3……a_{n-1}, a_n）

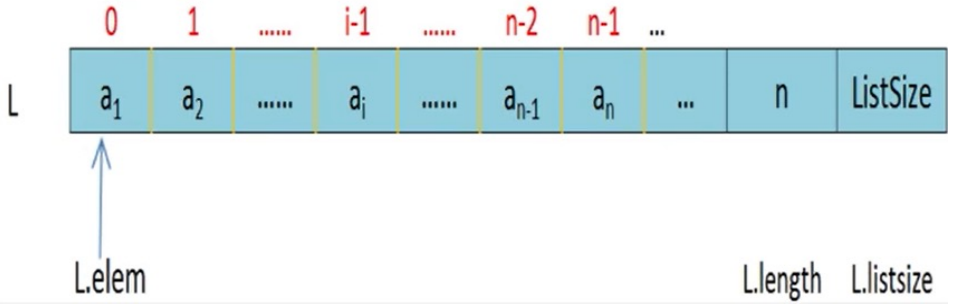
a_1称为栈底元素　　a_n称为栈顶元素

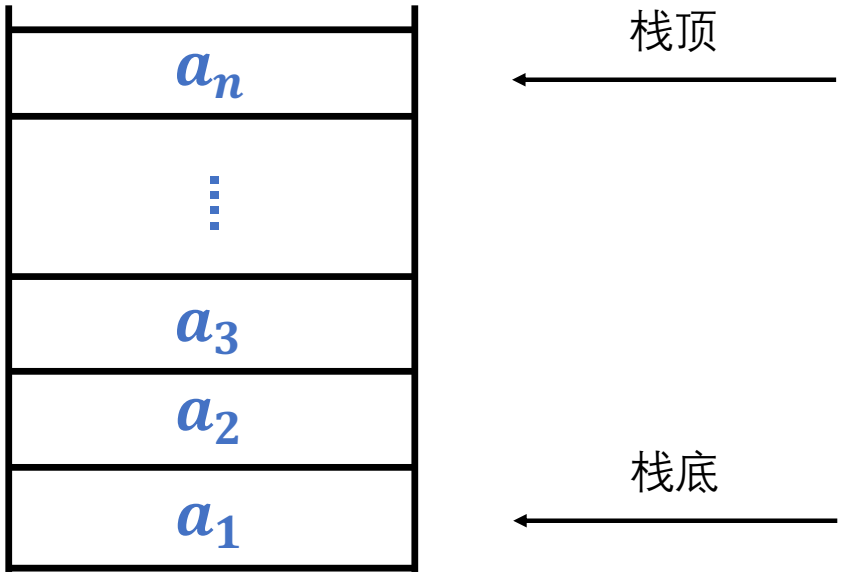
- 插入元素到栈顶（即表尾）的操作，称入栈。
- 从栈顶（即表尾）删除最后一个元素的操作，称为出栈。

“入”＝ 压入 ＝PUSH（x）　　“出”＝ 弹出 ＝POP（y）

3.1.1 栈的定义和特点

- 栈的示意图

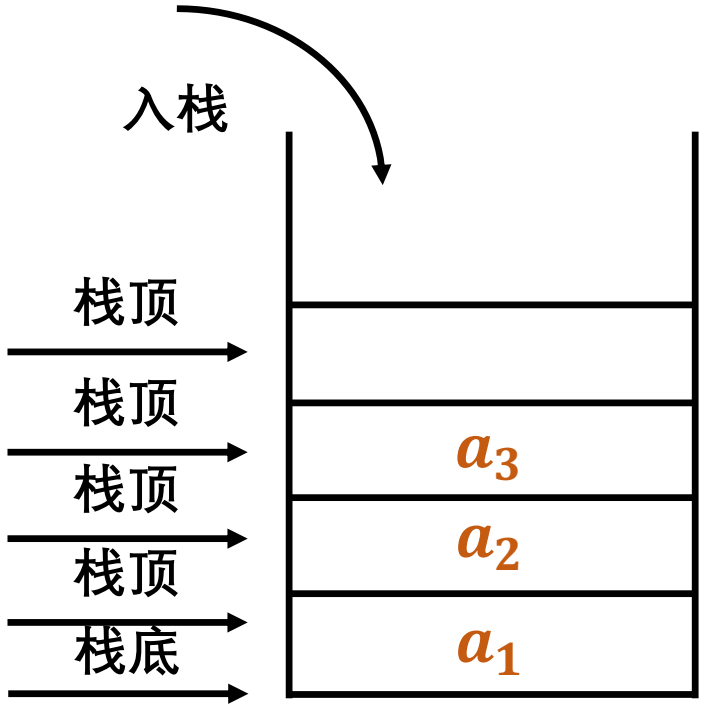




3.1.1 栈的定义和特点

杭州电子科技大学 HANGZHOU DIANZI UNIVERSITY

□ 入栈的操作示意图：

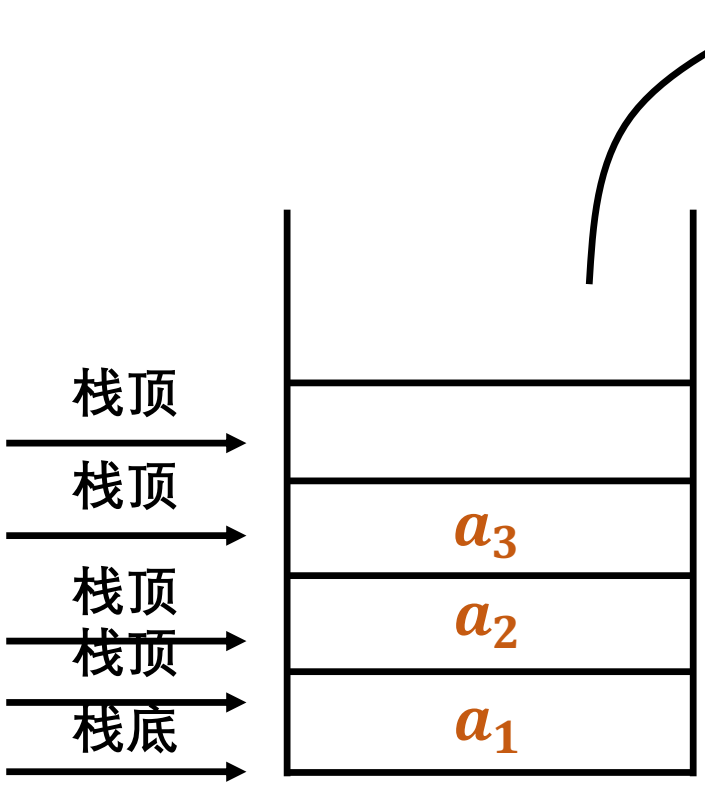


插入：入栈、进栈、压栈

3.1.1 栈的定义和特点

杭州电子科技大学 HANGZHOU DIANZI UNIVERSITY

- 出栈的操作示意图：



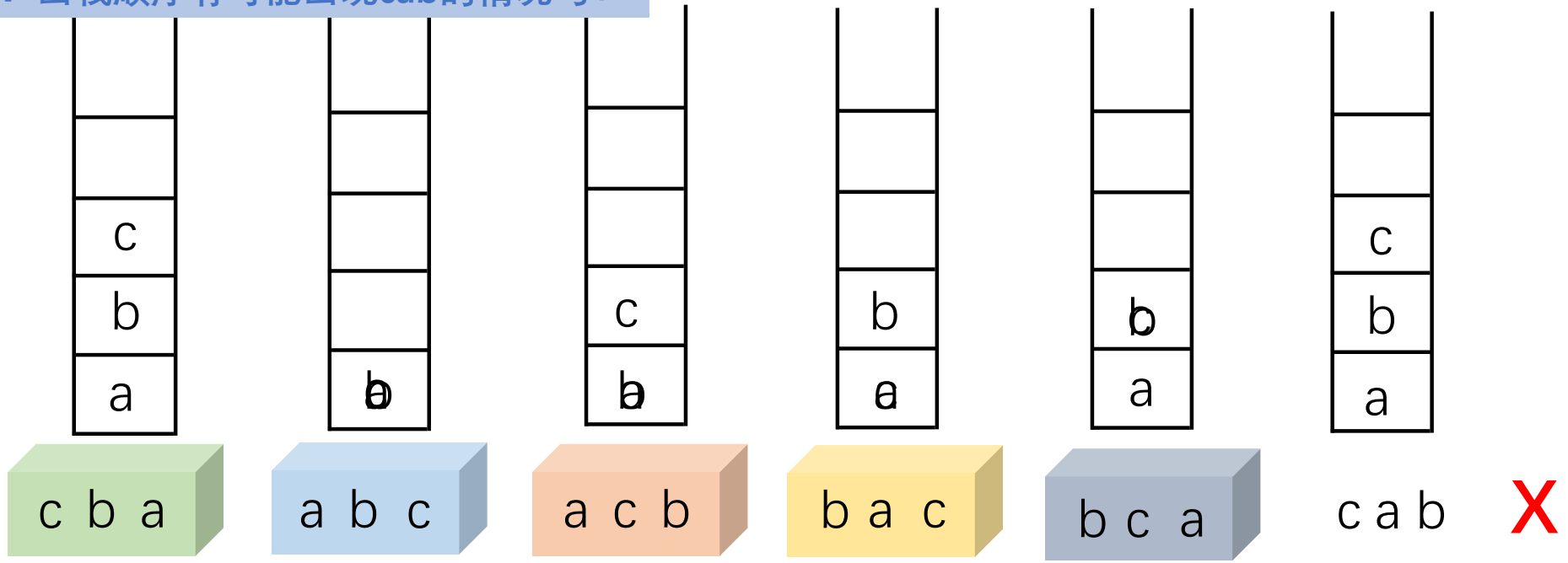
删除：出栈、弹栈

栈的操作特性：后进先出

3.1.1 栈的定义和特点

- [思考] 假设有3个元素a,b,c, 入栈的顺序是a,b,c 则它们出栈的顺序有几种可能？

思考：出栈顺序有可能出现cab的情况吗？



3.1.1 栈的定义和特点

栈的相关概念

1.定义　　限定只能在表的一端进行插入和删除运算的线性表（只能在栈顶操作）

2.逻辑结构　与线性表相同，仍为一对一关系

3.存储结构　用顺序栈或链栈存储均可，但以顺序栈更常见

4.运算规则　只能在栈顶运算，且访问节点时依照后进先出（LIFO）的原则

5.实现方式　关键是编写入栈和出栈函数，具体实现依顺序栈或链栈的不同而不同

3.1.1 栈的定义和特点

□ 栈与一般线性表有什么不同

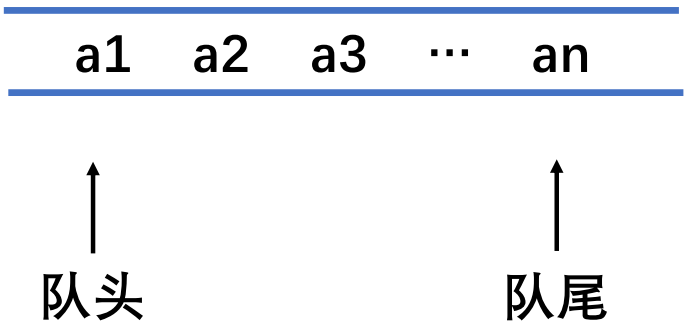
栈与一般线性表的区别：仅在运算规则不同。

一般线性表	栈
逻辑结构：一对一	逻辑结构：一对一
存储结构：顺序表、链表	存储结构：顺序栈、链栈
运算规则：可操作任意元素	运算规则：后进先出（LIFO）

3.1.2 队列的定义和特点

□ 队列(queue)是一种先进先出(First In First Out ---FIFO)的线性表。在表一端插入（表尾），在另一端（表头）删除。

$$Q = (a_1, a_2, \cdots a_n)$$



3.1.2 队列的定义和特点

□ 队列的相关概念

1.定义　只能在表的一端进行插入运算，在表的另一端进行删除运算的线性表（头删尾插）

2.逻辑结构　与线性表相同，仍为一对一关系

3.存储结构　顺序队或链队，但以循环顺序队列更常见

4.运算规则　只能在队首和队尾运算，且访问节点时依照先进先出（FIFO）的原则

5.实现方式　关键是掌握入队和出队操作，具体实现依顺序队或链队的不同而不同

课堂练习

1、若入栈顺序为A、B、C、D、E，则下列(　)出栈序列是不可能的。

- (A) . A、B、C、D、E
- (B) . B、C、D、A、E
- (C) . C、D、B、E、A
- (D) . D、E、C、A、B

第3章 栈和队列

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

3.1 栈和队列的定义和特点

3.2 栈的表示和操作的实现

3.3 栈的案例分析

3.4 栈和递归

3.5 队列的表示和操作的实现

3.2.1：栈的抽象数据类型的类型定义

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

ADT Stack {

数据对象：

D = { ai | ai ∈ ElemSet , i = 1,2,⋯,n, n ≥0 }

数据关系：

R1 = { $< a_{i-1}, a_i >$ | $a_{i-1}, a_i \in$ D, I = 2,⋯,n }

约定a_n端为栈顶，a_1端为栈底

基本操作：初始化、进栈、出栈、取栈顶元素等

} ADT Stack

3.2.1：栈的抽象数据类型的类型定义

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

InitStack(&S) 初始化操作
操作结果：构造一个空栈S

DestoryStack(&S) 销毁栈操作
初始条件：栈S已存在
操作结果：栈S被销毁

StackEmpty(S) 判定栈是否为空栈
初始条件：栈S已存在
操作结果：若栈S为空栈，则返回*True*，否则返回*False*

StackLength(S) 求栈的长度
初始条件：栈S已存在
操作结果：返回S的元素个数，即栈的长度

3.2.1：栈的抽象数据类型的类型定义

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

GetTop(S，&e) 取栈顶元素
初始条件：栈S已存在
操作结果：用e返回S的栈顶元素
ClearStack(&S) 栈置空操作
初始条件：栈S已存在
操作结果：将S清为空栈
Push(&S, e) 入栈操作
初始条件：栈S已存在
操作结果：插入元素e为新的栈顶元素

Pop(&S, &e) 出栈操作
初始条件：栈S已存在且非空
操作结果：删除S的栈顶元素a_n，并用e返回其值

栈的表示和实现

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

- 由于栈本身就是线性表，于是栈也有顺序存储和链式存储两种实现方式
  - 栈的顺序存储---顺序栈
  - 栈的链式存储---链栈

3.2.2： 顺序栈的表示和实现

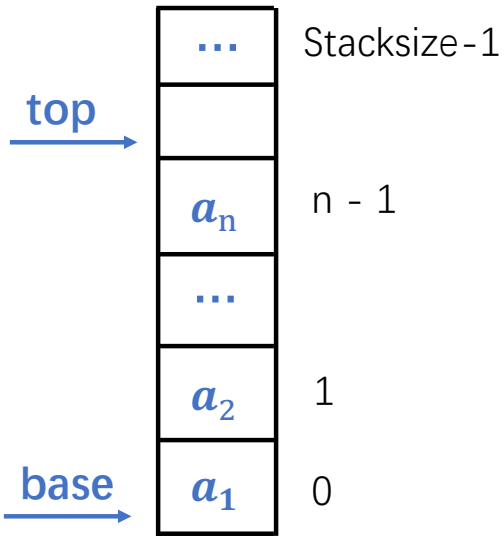
存储方式：同一般线性表的顺序存储结构完全相同

利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素。栈底一般在低地址端。

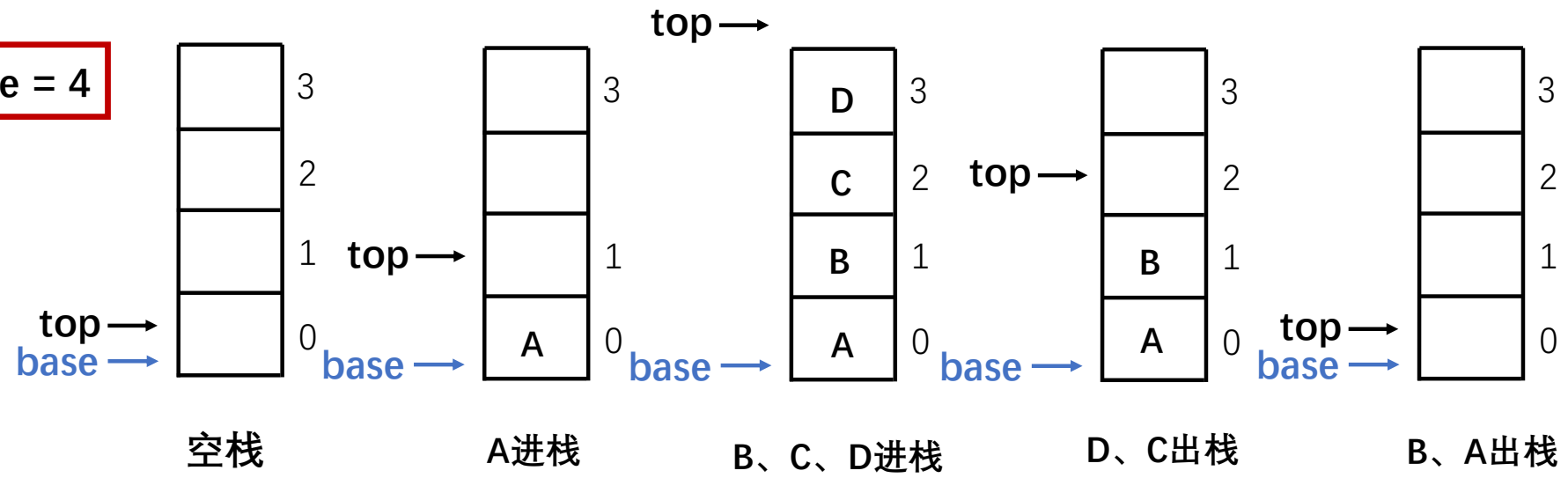
- 附设**top**指针，指示栈顶元素在顺序栈中的位置。
- 另设**base**指针，指示栈底元素在顺序栈中的位置。

但是，为了方便操作，通常top指示真正的栈顶元素之上的下标地址

- 另外，用stacksize表示栈可使用的最大容量



3.2.2： 顺序栈的表示和实现



空栈：base = top 是栈空标志

栈满: top – base == stacksize

栈满时的处理方法：
1、报错，返回操作系统
2、分配更大的空间，作为栈的存储控件，将原栈的内容移入新栈

3.2.2： 顺序栈的表示和实现

使用数组作为顺序栈存储方式的特点：

简单、方便、但易产生溢出（数组大小固定)

- 上溢（overflow）：栈已满，又要压入元素
- 下溢（underflow）：栈已经空，还要弹出元素

注：上溢是一种错误，使问题的处理无法进行；而下溢一般认为是一种结束条件，即问题处理结束。

3.2.2： 顺序栈的表示和实现

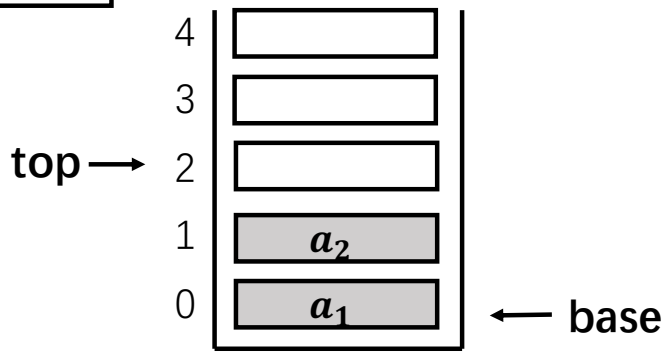
顺序栈的表示

```
#define  MAXSIZE  100
typedef struct{
        SElemType   *base; //栈底指针
        SElemType   *top;  //栈顶指针
        int stacksize;  //栈可用最大容量
}SqStack;
```

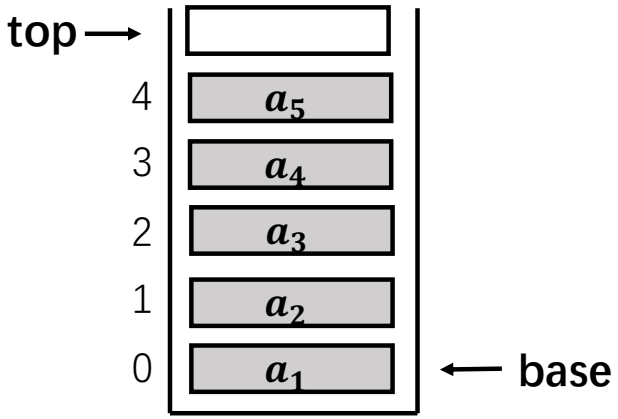
3.2.2： 顺序栈的表示和实现

顺序栈的表示

即stacksize -1



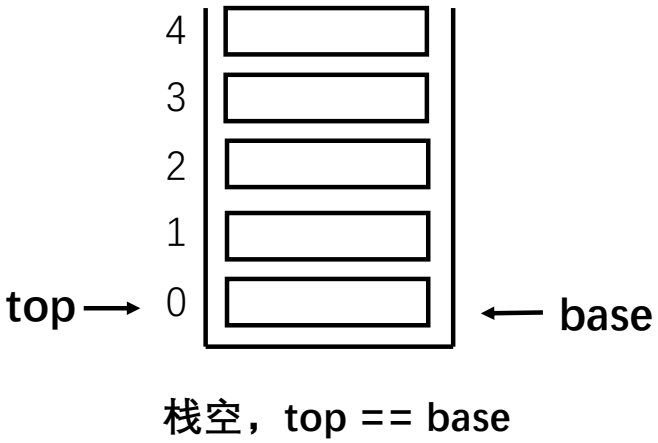
Stacksize:5
Top:2
Base:0
栈中元素个数 = top – base = 2



Stacksize:5
Top:5
Base:0
栈满：
栈中元素个数 = top – base = 5

3.2.2： 顺序栈的表示和实现

[算法3.1] 顺序栈的初始化



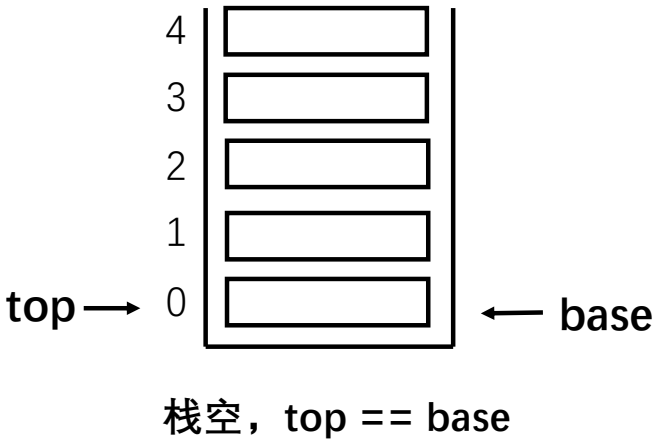
栈空，top == base

```
Status InitStack(SqStack   &S){   // 构造一个空栈
    S.base = new SElemType[MAXSIZE]; //或
    S.base = (SElemType*)malloc(MAXSIZE*sizeof(SElemType));
    if  (!S.base) exit  (OVERFLOW);  // 存储分配失败
    S.top = S.base;      //栈顶指针等于栈底指针
    S.stacksize = MAXSIZE;
    return OK;
}
```

3.2.2： 顺序栈的表示和实现

杭州电子科技大学 HANGZHOU DIANZI UNIVERSITY

[算法补充] 顺序栈是否为空

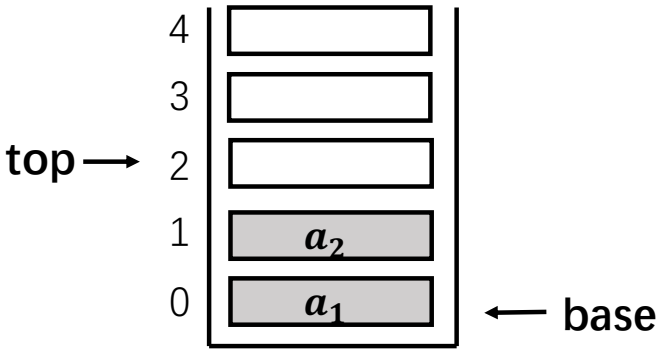


栈空，top == base

```
Status StackEmpty(SqStack S){
                // 若栈为空，返回TRUE；否则返回FALSE
    if (S.top == S.base)
        return TRUE;
    else
        return  FALSE;
}
```

3.2.2： 顺序栈的表示和实现

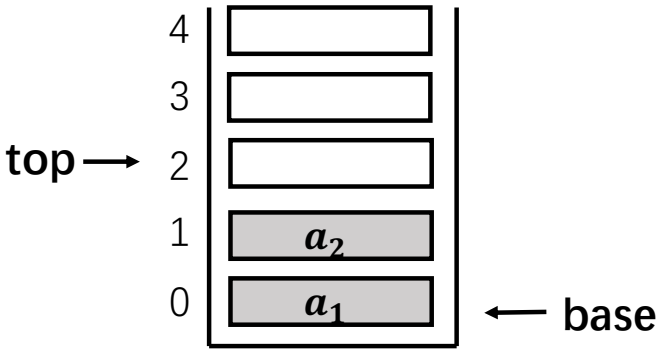
[算法补充] 求顺序栈长度




```
int StackLength( SqStack S )
{
        return S.top – S.base;
}
```

3.2.2： 顺序栈的表示和实现

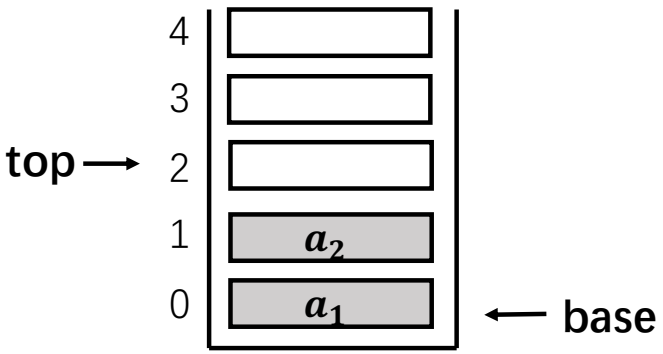
[算法补充] 清空顺序栈




```
Status ClearStack( SqStack S ) {
        if( S.base ) S.top = S.base;
        return OK;
}
```

3.2.2： 顺序栈的表示和实现

[算法补充] 销毁顺序栈

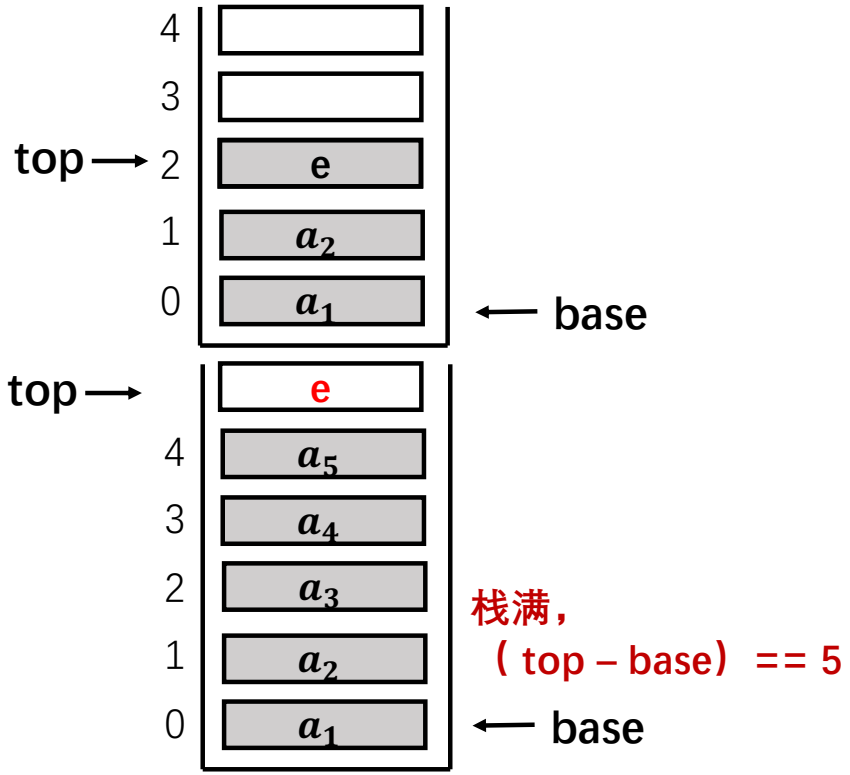



```
Status DestroyStack( SqStack &S ) {
    if( S.base ) {
        delete S.base ;
        S.stacksize = 0;
        S.base = S.top = NULL;
    }
    return OK;
}
```

3.2.2： 顺序栈的表示和实现

杭州电子科技大学 HANGZHOU DIANZI UNIVERSITY

[算法3.2] 顺序栈的入栈



（1）判断是否栈满，若满则出错（上溢）
（2）元素e压入栈顶
（3）栈顶指针加1

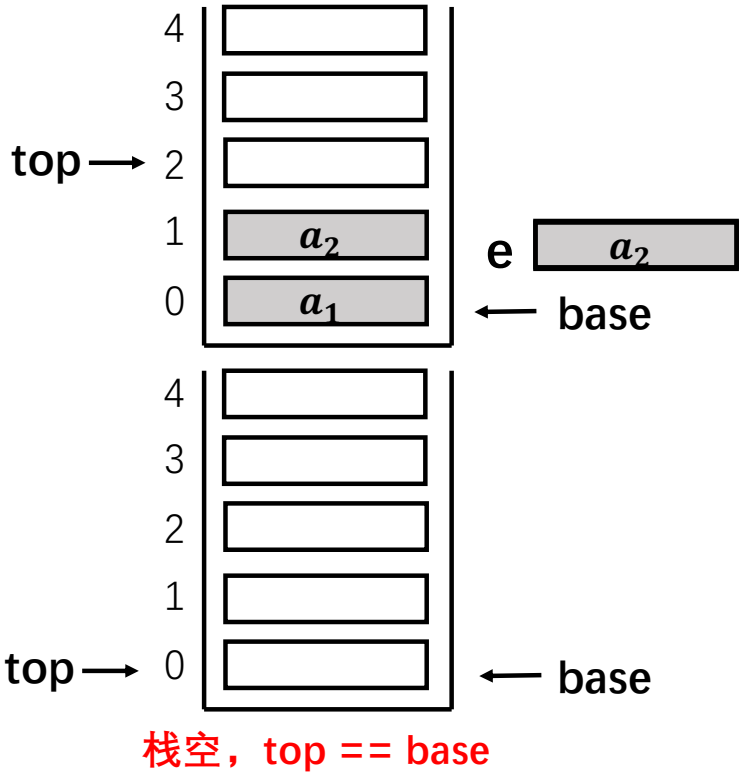
```
Status Push( SqStack &S, SElemType e) {
    if( S.top - S.base== S.stacksize ) // 栈满
        return ERROR;
    *S.top++=e;
    return OK;
}
```

```
*S.top=e;
S.top++;
```

3.2.2： 顺序栈的表示和实现

杭州电子科技大学 HANGZHOU DIANZI UNIVERSITY

[算法3.3] 顺序栈的出栈



栈空，top == base

（1）判断是否栈空，若空则出错（下溢）
（2）栈顶指针减1
（3）获取栈顶元素e

```
Status Pop(SqStack &S, SElemType &e) {
// 若栈不空，则删除S的栈顶元素，用e返回其值，并返回OK；
否则返回ERROR
    if(S.top == S.base)  // 等价于 if(StackEmpty(S))
        return ERROR;
    e = *--S.top;
    return OK;
}
```

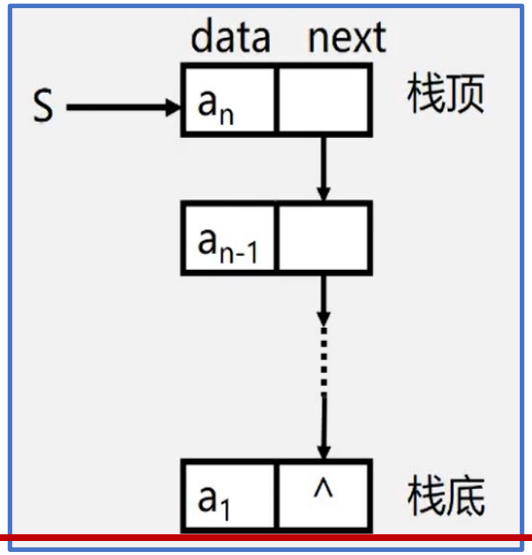
```
--S.top;
e=*S.top;
```

3.2.3： 链栈的表示和实现

链栈的表示

链栈是运算受限的单链表，只能在链表头部进行操作

```
typedef  struct StackNode{
      SElemType  data;
      struct StackNode *next;
 } StackNode,  *LinkStack;
LinkStack S;
```



- 链表的头指针就是栈顶
- 不需要头结点
- 基本不存在栈满的情况
- 空栈相当于头指针指向空
- 插入和删除仅在栈顶处执行

注意：链栈中指针的方向

3.2.3： 链栈的表示和实现

[算法3.4] 链栈的初始化

```
void InitStack(LinkStack &S ) {
    //构造一个空栈，栈顶指针置为空
   S=NULL;
   return OK;
}
```

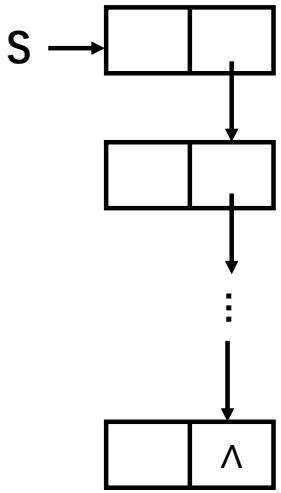
3.2.3： 链栈的表示和实现

[算法补充] 判断链栈是否为空

```
Status StackEmpty(LinkStack S)
    if (S==NULL) return TRUE;
    else return FALSE;
}
```

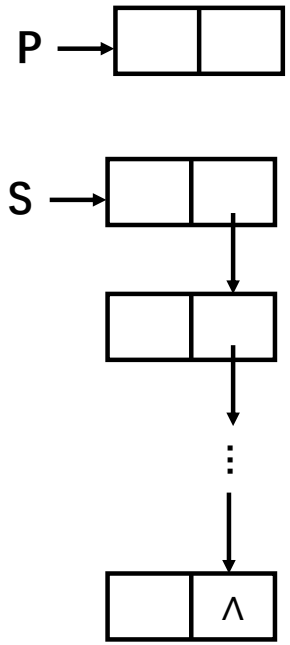
3.2.3： 链栈的表示和实现

[算法3.5] 链栈的入栈



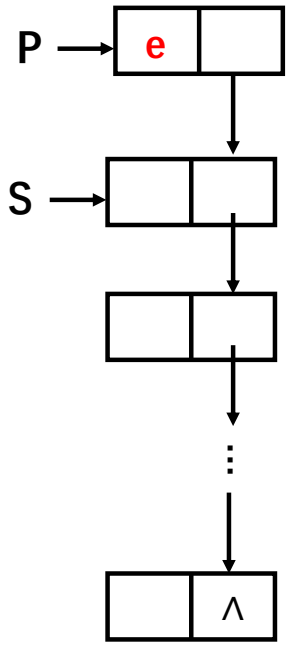
3.2.3： 链栈的表示和实现

[算法3.5] 链栈的入栈



3.2.3： 链栈的表示和实现

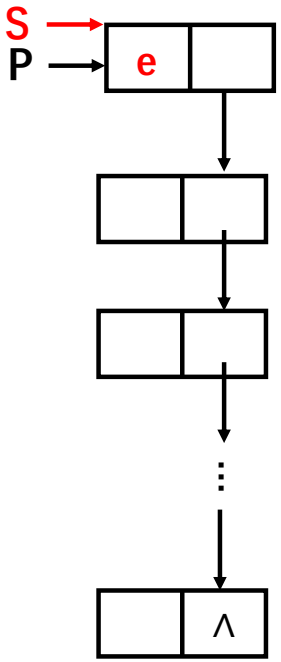
[算法3.5] 链栈的入栈




```
p->next = S
```

3.2.3： 链栈的表示和实现

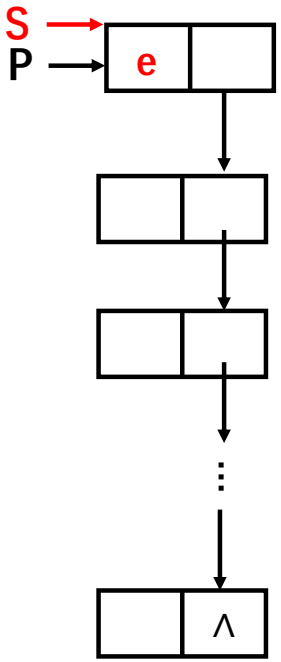
[算法3.5] 链栈的入栈



S = p

3.2.3： 链栈的表示和实现

[算法3.5] 链栈的入栈

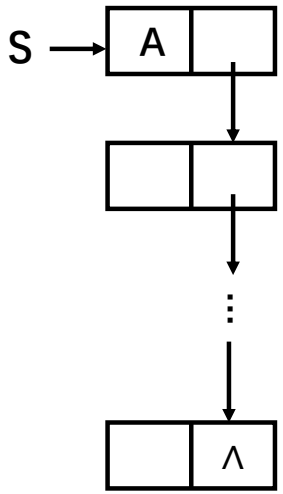



```
Status Push(LinkStack &S , SElemType e){
    p=new StackNode;      //生成新结点p
    p->data=e; //将新结点数据域置为e
    p->next=S;  //将新结点插入栈顶
    S=p;  //修改栈顶指针
    return OK;
}
```

3.2.3： 链栈的表示和实现

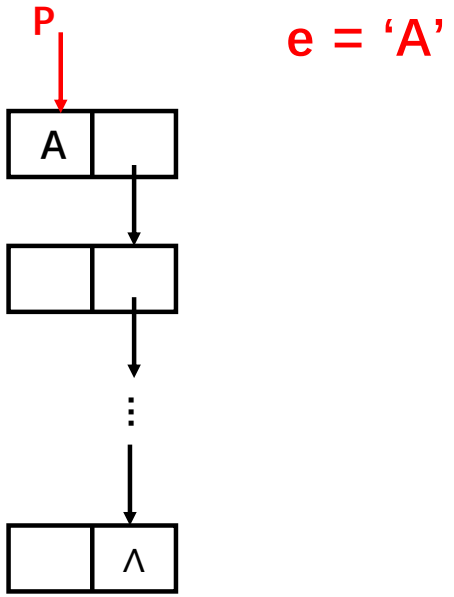
[算法3.6] 链栈的出栈

e = ‘A’



3.2.3： 链栈的表示和实现

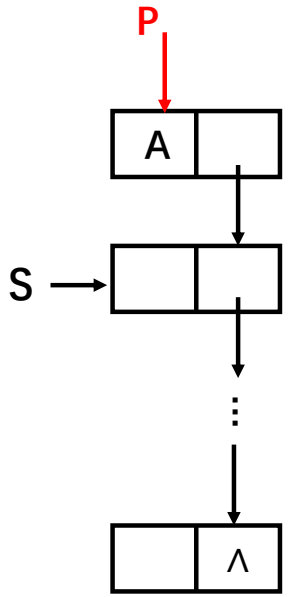
[算法3.6] 链栈的出栈



e = ‘A’

3.2.3： 链栈的表示和实现

[算法3.6] 链栈的出栈

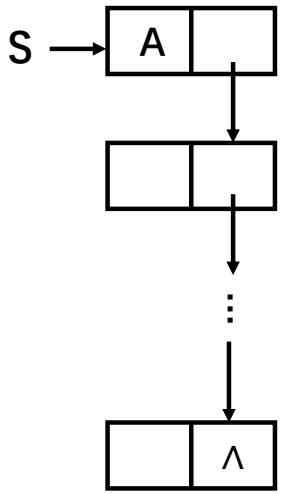


e = ‘A’

```
Status Pop (LinkStack &S,SElemType &e){
  if (S==NULL) return ERROR;
  e = S-> data;
  p = S;
  S =  S-> next;
  delete p;
  return OK;
}
```

3.2.3： 链栈的表示和实现

[算法3.7] 取栈顶元素




```
SElemType GetTop(LinkStack S) {
    if (S!=NULL)
        return S->data;
    }
```

第3章 栈和队列

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

3.1 栈和队列的定义和特点

3.2 栈的表示和操作的实现

3.3 栈的案例分析

3.4 栈和递归

3.5 队列的表示和操作的实现

3.3 栈的案例分析

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

案例1：进制转换

案例2：括号匹配的检验

案例3：行编辑程序

案例4：表达式求值

案例5：迷宫问题

案例1： 进制转换

- 十进制整数N向其他进制数d（二、八、十六）的转换是计算机实现计算的基本问题

  转换法则：除以d倒取余

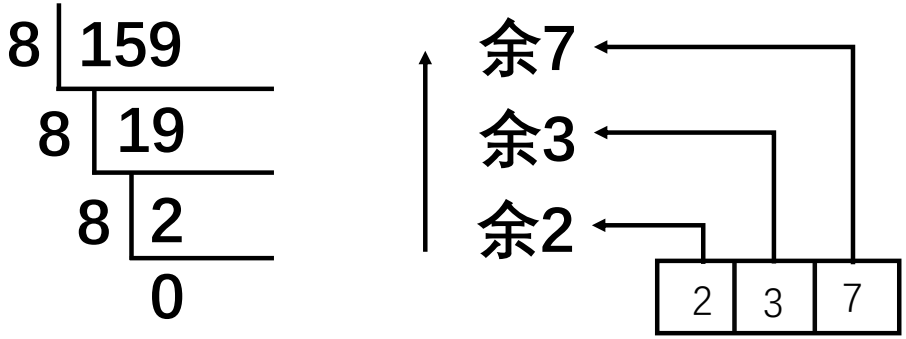
  该转换法则对应于一个简单算法原理：

  $$n = (n \text{ div } d) * d + n \bmod d$$

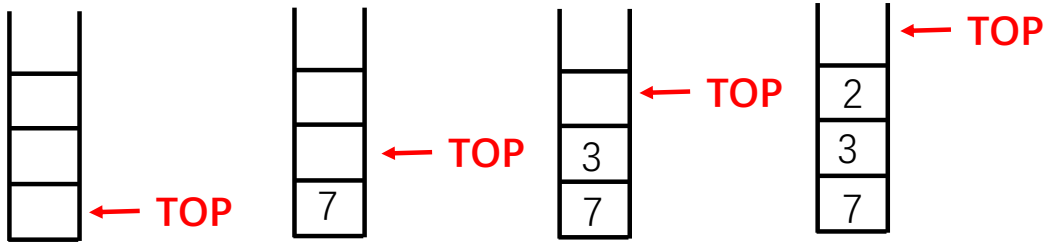
  其中：div 为整数运算，mod为求余运算

案例1： 进制转换

- 例 把十进制数159转换成八进制数



$$(159)_{10} = (237)_8$$



案例1： 进制转换


```
void conversion ( ) {
    // 对于输入的任意一个非负十进制整数，打印输出与其等值的八进制数
    int e;  SqStack S;
    InitStack(S);                    //构造空栈
    scanf("%d", N)
    while (N) {
        Push(S, N % 8);  N = N/8;
    }
    while (!StackEmpty(S)) {
        Pop(S,e);
        printf ("%d", e);
    }
} //conversion
```

案例2： 括号匹配的检验

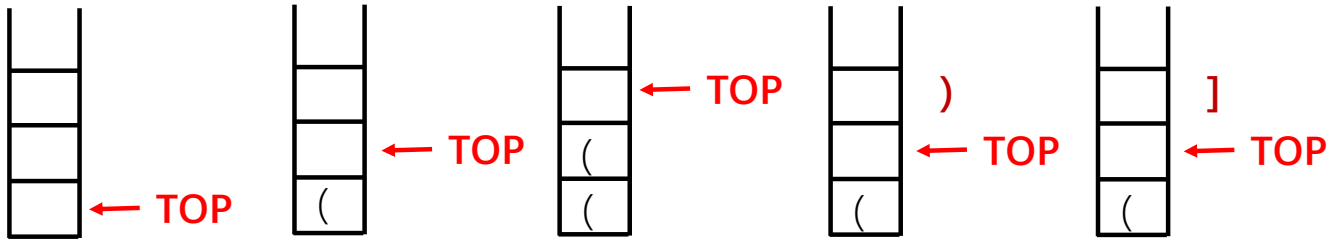
- 假设表达式中允许包含两种括号：圆括号和方括号
  其嵌套的顺序随意，即：

  1. ([ ] ( )) 或 [([ ] [ ])] 为正确格式;

  2. [( ]) 为错误格式;

  3. ([( )) 或 (( )]) 为错误格式;

例如：检验 (( )]) 是否匹配



案例2： 括号匹配的检验

- 可以利用一个栈结构保存每个出现的左括号，当遇到右括号时，从栈中弹出左括号，检验匹配情况。
- 从检验过程中，若遇到以下几种情况之一，就可以得出括号不匹配的结论。

(1) 当遇到一个右括号时，栈已空，说明目前为止，右括号多余左括号；

(2) 从栈中弹出的左括号与当前检验的右括号类型不同，说明出现了括号交叉的情况；

(3) 算术表达式输入完毕，但栈中还有没有匹配的左括号，说明左括号多余右括号；

案例2： 括号匹配的检验

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

```
void Match_Brackets( ){
    SqStack S;  InitStack(S);  // 初始化
    char ch, x;
    scanf("%c", &ch) ;
    while (ch!='\n'){
        //通过栈匹配括号
        scanf("%c", &ch) ;
    }
    if(!StackEmpty(S)){
        printf( “括号数量不匹配！” ) ;
        return FLASE;
    }
    else  return TRUE;
}
```

```
if((ch == '(') || (ch == '[')) Push(S, ch);
else if(ch == ']'){
        Pop(S, x);
        if(x != '['){
            printf( “'['括号不匹配” ) ;
            return FLASE;
        }
    }
else if(ch == ')'){
        Pop(S, x);
        if(x != '('){
            printf( “'('括号不匹配” ) ;
            return FLASE;
        }
    }
```

案例3： 行编辑程序

一个简单的行编辑程序的功能是：接受用户从终端输入的程序或数据，并存入用户的数据区。由于用户在终端上进行输入时，不能保证不出差错，因此，若在编辑程序中，“每接受一个字符即存入用户数据区”的做法显然不是最恰当的。

较好的做法是：在编辑程序中，设立一个输入缓冲区，用于接受用户输入的一行字符，然后逐行存入用户数据区。允许用户输入错误，并在发现有误时可以及时更正。

例如：当用户发现刚刚键入的一个字符是错的时，可补进一个退格符“#”，以表示前一个字符无效；如果发现当前键入的行内差错较多或难以补救，则可以键入一个退格符“@”，以表示当前行中的字符均无效。

例如：假设从终端接受了这样两行字符：

```
whli##ilr#e(s#*s)
  outcha@putchar(*s=#++);
```

则实际有效的是下列两行：

```
while(*s)
  putchar(*s++);
```

案例3： 行编辑程序

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

```
void LineEdit() {
  SqStack S;
  InitStack(S);       //构造空栈S
  ch = getchar();     //从终端接收第一个字符
  while (ch != EOF) { //EOF为全文结束符

        ……

  }
  DestroyStack(S);
}
```

```
while (ch != EOF && ch != '\n') {
    switch (ch) {
        case '#': Pop(S, ch);    break;  // 仅当栈非空时退栈
        case '@': ClearStack(S); break;  // 重置S为空栈
        default : Push(S, ch);   break;  // 有效字符进栈,未考
虑栈满情形
    }
    ch = getchar();  // 从终端接收下一个字符
}
// 将从栈底到栈顶的栈内字符传送至调用过程的数据区;
ClearStack(S);      // 重置S为空栈
if (ch != EOF) {
    ch = getchar();
}
```

案例4： 表达式求值

- 表达式求值是程序设计语言编译中一个最基本的问题，它的实现也需要用到栈。

- 这里介绍的算法是由算法优先级确定运算顺序的对表达式求值算法

——算符优先算法

例如：算术表达式4＋2×3－10／5 的计算顺序为：

4＋2×3－10／5 ＝4 ＋6 －10 ／5 ＝10 －10 ／5 ＝10 －2 ＝8

案例4： 表达式求值

- 表达式的组成
  - 操作数（operand）： 常数，变量
  - 运算符（operator）： 算术运算符、关系运算符和逻辑运算符
  - 界限符（delimiter）： 左右括弧和表达式结束符

- 任何一个**算术表达式**都由操作数（常数、变量）、算术运算符（+、-、*、/）和界限符（括号、表达式结束符'#'、虚设的表达式起始符'#'）组成。后两者统称为算符。

  例如：#3*（7－2）#

案例4： 表达式求值

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

为了实现表达式求值，需要设置两个栈：

一个是算符栈OPTR，用于寄存运算符
另一个称为操作数栈OPND，用于寄存操作数和运算结果

求值的处理过程自左至右扫描表达式的每一个字符

- 当扫描到的是操作数，则将其压入栈OPND
- 当扫描到的是运算符时
  - 若这个运算符比OPTR栈顶运算符的优先级高，则入栈OPTR，继续向后处理
  - 若这个运算符比OPTR栈顶运算符优先级低，则从OPND栈中弹出两个运算数，从栈OPTR中弹出栈顶运算符进行运算，并将运算结果压入栈OPND
- 继续处理当前字符，直到遇到结束符为止。

案例4： 表达式求值

算符间优先级关系

表 3.1 算符间的优先关系

θ_1 \ θ_2	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

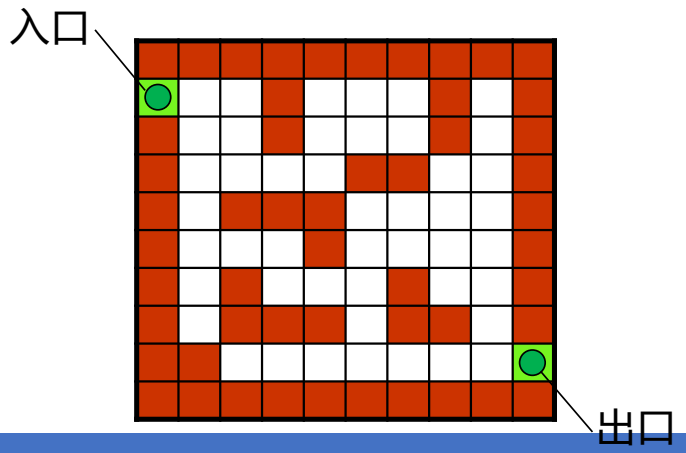
案例4

```
OperandType  EvaluateExpression() {
  // 算术表达式求值的算符优先算法。设 OPTR 和 OPND 分别为运算符栈和运算数栈，
  // OP 为运算符集合。
   InitStack (OPTR);   Push (OPTR,'#');
   initStack (OPND);   c = getchar();
   while (c!='#' || GetTop(OPTR)!='#') {
     if (!In(c, OP)){Push((OPND, c); c = getchar(); }     // 不是运算符则进栈
     else
       switch (Precede(GetTop(OPTR), c)) {
         case '<':   // 栈顶元素优先权低
                Push(OPTR, c);  c = getchar();
                break;
         case '=':   // 脱括号并接收下一字符
                Pop(OPTR, x);   c = getchar();
                break;
         case '>':   // 退栈并将运算结果入栈
               Pop(OPTR, theta);
               Pop(OPND, b);  Pop(OPND, a);
               Push(OPND, Operate(a, theta, b));
               break;
         }// switch
   }// while
   return GetTop(OPND);
}// EvaluateExpression
```

案例5： 迷宫问题

求迷宫中从入口到出口的所有路径是一个经典的程序设计问题。计算机解迷宫时，通常用的是“穷举求解”的方法：即从入口出发，顺某一方向向前探索，若能走通，则继续往前走；否则沿原路退回，换一个方向再继续探索，直至所有可能的通路都探索到为止。

为了保证在任何位置上都能沿原路退回，显然需要用一个后进先出的结构保存从入口到当前位置的路径。因此，在求迷宫通路的算法中应用“栈”也就是自然而然的事了。



分析：假设“当前位置”指的是“在搜索过程中某一时刻所在图中某个方块位置”，则求迷宫中一条路径的算法的基本思想是：

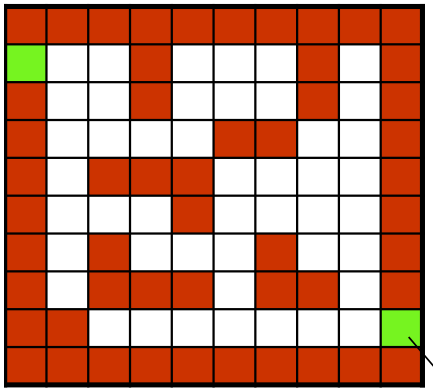
- 若当前位置“可通”，则纳入“当前路径”，并继续朝“下一位置”探索，即切换“下一位置”为“当前位置”，如此重复直至到达出口；
- 若当前位置“不可通”，则应顺着“来向”退回到“前一通道块”，然后朝着除“来向”之外的其他方向继续探索；
- 若该通道块的四周4个方块均“不可通”，则应从“当前路径”上删除该通道块。

案例5： 迷宫问题


```
设定当前位置的初值为入口位置;
do{
    若当前位置可通,
    则{
        将当前位置插入栈顶;
        若该位置是出口位置，则结束;
        否则，切换当前位置的“东”邻方块为新的当前位置;
    }
    否则{
        若栈不空且栈顶位置尚有其他方向未经探索,
        则 设定新的当前位置为沿顺时针方向旋转找到的栈顶位置的下一相邻块;
        若栈不空但栈顶位置的四周均不可通,
        则{
            删去栈顶位置;
            若栈不空，则重新测试新的栈顶位置，直至找到一个可通的相邻块或出栈至栈空;
        }
    }
}while(栈不空);
```

所谓当前位置**可通**，指的是**未曾走到过的通道块**，即要求该方块位置不仅是通道块，而且既不在当前路径上（否则所求路径就不是简单路径），也不是曾经纳入过路径的通道块（否则只能在死胡同内转圈）。

案例5：




```
typedef struct {
  int         ord;          // 通道块在路径上的"序号"
  PosType     seat;         // 通道块在迷宫中的"坐标位置"
  int         di;           // 从此通道块走向下一通道块的"方向"
}SElemType;                 // 栈的元素类型

Status MazePath ( MazeType maze, PosType start, PosType end ) {
  // 若迷宫 maze 中存在从入口 start 到出口 end 的通道,则求得一条存放在栈中(从栈底到栈
  // 顶),并返回 TRUE;否则返回 FALSE
    InitStack(S);  curpos = start;        // 设定"当前位置"为"入口位置"
    curstep = 1;                          // 探索第一步
    do {
      if (Pass (curpos)) {  // 当前位置可以通过,即是未曾走到过的通道块
          FootPrint (curpos);                 // 留下足迹
          e = ( curstep, curpos, 1 );
          Push (S,e);                         // 加入路径
          if (curpos == end)  return (TRUE);  // 到达终点(出口)
          curpos = NextPos ( curpos, 1 );     // 下一位置是当前位置的东邻
          curstep++ ;                         // 探索下一步
        }// if
        else {  // 当前位置不能通过
        if (!StackEmpty(S)) {
          Pop (S,e);
          while (e.di == 4 && !StackEmpty(S)) {
            MarkPrint (e.seat);  Pop (S,e);       // 留下不能通过的标记,并退回一步
          }// while
          if (e.di<4) {
            e.di++ ;   Push ( S, e);              // 换下一个方向探索
            curpos = NextPos (e.seat e.di );      // 设定当前位置是该新方向上的相邻块
           }// if
         }// if
       }// else
     }while ( !StackEmpty(S) );
   return (FALSE);
}// MazePath
```